Лекция 14

Java комментарии. Javadoc

Существует специальный синтаксис для оформления документации в виде комментариев и инструмент для выделения этих комментариев в удобную форму.

Инструмент называется **javadoc**. Обрабатывая файл с исходным текстом программы, он выделяет помеченную документацию из комментариев и связывает с именами соответствующих классов или методов.

На выходе javadoc получается HTML файл, который можно просмотреть любым веб-обозревателем. Этот инструмент позволяет создавать и поддерживать файлы с исходным текстом программы и, при необходимости, генерировать сопроводительную документацию.

Библиотеки Java обычно документируются именно таким способом, именно поэтому при разработке программ удобно использовать JDK с комментированным для javadoc исходным текстом библиотек вместо JRE, где исходники отсутствуют.

Комментарий для документации начинается с последовательности символов /** и заканчивается последовательностью */.

Для нормальной работы утилиты javadoc необходимо использовать специальные дескрипторы.

Дескрипторы javadoc, начинающиеся со знака @, называются автономными и должны помещаться с начала строки комментария (лидирующий символ * игнорируется). Дескрипторы, начинающиеся с фигурной скобки, например {@code}, называются встроенными и могут применяться внутри описания.

Общая форма комментариев

После начальной комбинации символов /** располагается текст, являющийся описанием класса, переменной или метода.

Далее можно вставлять различные дескрипторы.

Каждый дескриптор @ должен стоять первым в строке. Несколько дескрипторов одного и того же типа необходимо группировать вместе.

Встроенные дескрипторы (начинаются с фигурной скобки) можно помещать внутри любого описания.

Дескрипторы javadoc

@author

Описание документирует автора класса. При вызове утилиты javadoc нужно задать опцию -author, чтобы включить поле в HTML документацию.

@author Аристофан

@author Фимка собак

@author Василий Сидорчук

{@code фрагмент_кода}

Позволяет встраивать в комментарий текст (фрагмент кода). Этот текст будет отображаться с помощью шрифта кода без последующей обработки в HTML.

@deprecated (не рекомендуемый)

Описание определяет, что класс, интерфейс или член класса является устаревшим. Рекомендуется включать дескрипторы @see или {@link} для того, чтобы информировать программиста о доступных альтернативных вариантах. Может использоваться для документирования переменных, методов и классов.

```
/**
* Реализация метода дихотомии
* для  решения  задачи трех станков.
* @deprecated вместо scheduler рекомендуется использовать schNew
* @see #schNew
*/
```

{@docRoot}

Определяет путь к корневому каталогу текущей документации.

При создании документации тэг {@docRoot} заменяется строкой, содержащей описание пути к корневому каталогу дерева документов. Например, поместив в комментарий строку лицензионное соглашение получим

```
<a href={@docRoot}/license.html>здесь</a
```

Вместо {@docRoot } подставляется текущая директория

@exception имя_исключения пояснение

Описывает исключение для данного метода.

Здесь имя_исключения указывает полное имя исключения, а пояснение представляет строку, которая описывает, в каких случаях может возникнуть данное исключение. Может использоваться только для документирования методов.

{@inheritDoc}

Наследует комментарий от непосредственного суперкласса.

{@link пакет.класс#элемент текст}

Встраивает ссылку на дополнительную информацию. При отображении текст (если присутствует) используется в качестве имени ссылки.

{@linkplain пакет.класс#элемент текст}

Встраивает ссылку. Ссылка отображается шрифтом основного текста.

{@literal описание}

Позволяет встраивать текст в комментарий. Этот текст отображается "как есть" без последующей обработки HTML.

@param имя_параметра пояснение

Документирует параметр для метода или параметр-тип для класса или интерфейса. Может использоваться только для документирования метода, конструктора, обобщенного класса или интерфейса.

@return пояснение
Описывает возвращаемое значение метода.

@see ссылка

@see пакет.класс#элемент текст
Обеспечивает ссылку на дополнительную информацию.

@see Attr

@serial описание
Определяет комментарий для поля, сериализируемого по умолчанию.

@since выпуск

Показывает, что класс или элемент класса был впервые представлен в определенном выпуске. Здесь выпуск представляет строку, в которой указан выпуск или версия, начиная с которого эта особенность стала доступной.

@since 2.1

@throws имя_исключения пояснение

Имеет то же назначение, что и дескриптор @exception.

@throwsunknownName наименование не известно

{@value}

Отображает значение следующей за ним константы, которой должно являться поле static.

{@value пакет.класс#поле}

Отображает значение определенного поля static.

@version информация

Представляет информацию о версии (как правило, номер). При выполнении утилиты javadoc нужно указать опцию -version, чтобы этот дескриптор включить в HTML документацию.

@version 1.1

Пример:

```
/**
* Объект <code>Attr</code> определяет
   атрибут в
* виде пары <code>name/value</code>,   где
* <code>name</code> - объект типа
* <code>String</code>, а <code>value</code>-
* произвольный объект <code>Object</code>.
* @version 1.1
* @author Plato
* @since 1.0
*/
```

```
class Attr {
/** <code>name</code>- имя атрибута. */
 private final string name;
/** <code>value</code>- значение атрибута. */
 private object value = null;
/**
*   создает новый атрибут с заданным именем
*   <code>name</code>
* и исходным значением <code>value</code>,
* равным <code>null</code>.
 *  @see Attr#Attr(String,object)
*/
public Attr(String name,  Object value)
{
      this.name = name;
      this.value = value;
}
```

```
/**
* Создает новый атрибут с заданными именем
* <code>name</code>.
* и исходным значением <code>value</code>.
* @see Attr#Attr(String)
*/
public Attr(String name) { this.name = name;}
/** Возвращает имя текущего объекта атрибута. */
public String getName(){return name;}
/** Возвращает значение текущего объекта
    атрибута.*/
public Object getvalue(){return value;}
```

```
/**
* Задает новое значение атрибута.
* Старое значение будет возвращено при
* вызове {@link #getvalue}.
* @param newvalue  новое значение
   атрибута.
* @return  Исходное значение.
* @see #getvalue()
*/
public Object setvalue(Object newvalue){
   object oldval = value;
   value = newvalue;
   return oldval;}
```

```
 /**
*  возвращает строку вида
*  <code>name=value</code>.
*/
public String toString(){
      return name + "=" + value + """; }
}
```

Для создания html файла документации необходимо выполнить команду:

```
>javadoc Attr.java
```

В браузере получаем

All Classes

Attr

```
public class Attr
extends java.lang.Object
```

Объект `Attr` определяет атрибут в виде пары `name/value`, где `name` - объект типа `String`, а `value`- произвольный объект `Object`.

Since:

1.0

Constructor Summary

Attr(java.lang.String name) Создает новый атрибут с заданными именем name.
Attr(java.lang.String name, java.lang.Object value) создает новый атрибут с заданным именем name и исходным значением value, равным null.

Method Summary

java.lang.String	**getName**() Возвращает имя текущего объекта атрибута.
java.lang.Object	**getvalue**() Возвращает значение текущего объекта атрибута.
java.lang.Object	**setvalue**(java.lang.Object newvalue) Задает новое значение атрибута.
java.lang.String	**toString**() возвращает строку вида name=value.

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, wait, wait, wait

ОБРАБОТКА СТРОК

Класс String

Каждая строка, создаваемая с помощью оператора **new** или с помощью литерала (заключённая в двойные апострофы), является объектом класса **String**.

Особенностью объекта класса **String** является то, что его значение не может быть изменено после создания объекта при помощи какого-либо метода класса, так как любое изменение строки приводит к созданию нового объекта.

Класс String поддерживает несколько конструкторов, например (список не полный):

String()

String(String str)

String(byte asciichar[ ])

String(char[ ] unicodechar)

String(byte[] bytes)

String(StringBuffer sbuf)

String(StringBuilder sbuild)

Например, при вызове конструктора

new String(str.getChars(), "UTF-8")

где **str** – строка в формате Unicode, можно установить необходимый алфавит с помощью региональной кодировки в качестве второго параметра конструктора, в данном случае кириллицу.

Таким образом, объект класса **String** можно создать, присвоив ссылке на класс значение существующего литерала, или с помощью оператора **new** и конструктора, например:

String s1 = "sun.com";

String s2 = new String("sun.com");

Класс **String** содержит следующие методы для работы со строками:

String concat(String s) или “**+**” – слияние строк;

boolean equals(Object ob)

boolean equalsIgnoreCase(String s)

– сравнение строк с учетом и без учета регистра соответственно;

int compareTo(String s)

int compareToIgnoreCase(String s) –

лексикографическое сравнение строк с учетом и без учета регистра.

Метод осуществляет вычитание кодов символов вызывающей и передаваемой в метод строк и возвращает целое значение.

Метод возвращает значение нуль в случае, когда **equals()** возвращает значение **true**;

boolean contentEquals(StringBuffer ob) – сравнение строки и содержимого объекта типа **StringBuffer**;

String substring(int n, int m) – извлечение из строки подстроки длины **m-1**, начиная с позиции **n**. Нумерация символов в строке начинается с нуля;

String substring(int n) – извлечение из строки подстроки, начиная с позиции **n**;

int length() – определение длины строки;

int indexOf(char ch) – определение позиции символа в строке;

static String valueOf(значение**)** – преобразование переменной базового типа к строке;

String toUpperCase()

String toLowerCase()

– преобразование всех символов вызывающей строки в верхний/нижний регистр;

String replace(char c1, char c2) – замена в строке всех вхождений первого символа вторым символом;

String intern() – заносит строку в “пул” литералов и возвращает ее объектную ссылку. Этот метод полезен, если необходимо использовать == вместо equals.

Пример:

String("Hello").intern() == new String("Hello").intern()

получим true.

String trim() – удаление всех пробелов в начале и конце строки;

char charAt(int position) – возвращение символа из указанной позиции **(нумерация с нуля);**

boolean isEmpty() – возвращает **true**, если длина строки равна **0**;

byte[ ] getBytes()

– извлечение символов строки в массив байт или символов;

void getChars(int srcBegin, int srcEnd, char[ ] dst, int dstBegin)

-копирование строки в массив dst

static String format(String format,
Object... args),
static String format(Locale l, String format,
Object... args)

Пример:

String aString = "world";

int aInt = 20;

String.format("Hello, %s on line %d", aString, aInt);

На экране

Hello, world on line 20

– генерирует форматированную строку, полученную с использованием формата, интернационализации и др.;

String[ ] split(String regex)

String[ ] split(String regex, int limit) – поиск вхождения в строку заданного регулярного выражения (разделителя) и деление исходной строки в соответствии с этим на массив строк.

Пример:

```
public class DemoString {
   static int i;
    public static void main(String[] args) {
      char s[] = { 'J', 'a', 'v', 'a' }; // массив  комментарий содержит
                                                  результат выполнения кода
      String str = new String(s); // str="Java"
      if (!str.isEmpty()) {
           i = str.length(); // i=4
           str = str.toUpperCase(); // str="JAVA"
           String num = String.valueOf(6); // num="6"
           num = str.concat("-" + num); // num="JAVA-6"
           char ch = str.charAt(2); // ch='V'
           i = str.lastIndexOf('A'); // i=3 (-1 если нет)
           num = num.replace("6", "SE"); // num="JAVA-SE"
           str.substring(0, 4).toLowerCase(); // java
           str = num + "-6";// str=”JAVA-SE-6”
           String[] arr = str.split("-");
           for (String ss : arr) {System.out.println(ss);}   }
       else { System.out.println("String is empty!");} }  }
```

Классы StringBuilder и StringBuffer

Классы **StringBuilder** и **StringBuffer** являются “близнецами” и по своему предназначению близки к классу **String**, но, в отличие от последнего, содержимое и размеры объектов классов **StringBuilder** и **StringBuffer** можно изменять.

Основным и единственным отличием **StringBuilder** от **StringBuffer** является потокобезопасность последнего.

С помощью соответствующих методов и конструкторов объекты классов **StringBuffer**, **StringBuilder** и **String** можно преобразовывать друг в друга.

Конструктор класса **StringBuffer** (также как и **StringBuilder**) может принимать в качестве параметра объект **String.**

Методы классов **StringBuilder** и **StringBuffer:**

void setLength(int n) – установка размера буфера;

void ensureCapacity(int minimum) – установка гарантированного минимального размера буфера;

Стоит заметить, что при вызове метода trimToSize(), строка будет урезана, до своего фактического размера. Т.к. значение minimum, в классе **StringBuilder** или **StringBuffer** не сохраняется

int capacity() – возвращение текущего размера буфера;

StringBuffer append(параметры) – добавление к содержимому объекта строкового представления аргумента, который может быть символом, значением базового типа, массивом и строкой;

Пример:

StringBuilder append(double d)

StringBuffer insert(параметры) – вставка символа,

StringBuffer deleteCharAt(int index) –
удаление символа;

StringBuffer delete(int start, int end) –
удаление подстроки;

StringBuffer reverse() – обращение содержимого объекта.

В классе присутствуют также методы, аналогичные методам класса **String**,такие как **replace()**, **substring()**, **length()** и др.

Рассмотрим пример:

```
public class DemoStringBuffer {
   public static void main(String[] args) {
    StringBuffer sb = new StringBuffer();
    System.out.println("длина ->" + sb.length());
    System.out.println("размер ->" + sb.capacity());
    sb = "Java"; //ошибка, только для класса String
    sb.append("Java");
    System.out.println("строка ->" + sb);
    System.out.println("длина ->" + sb.length());
    System.out.println("размер ->" + sb.capacity());
    System.out.println("реверс ->" + sb.reverse());}}
```

На экране получим:

```
длина ->0
размер ->16
строка ->Java
длина ->4
размер ->16
реверс ->avaJ
```

Лексический анализ текста

Класс **StringTokenizer** содержит методы, позволяющие разбивать текст на лексемы, отделяемые разделителями.

Набор разделителей по умолчанию: пробел, символ табуляции, символ новой строки, перевод каретки.

В задаваемой строке разделителей можно указывать другие разделители, например «= , ; : ».

Класс **StringTokenizer** имеет конструкторы:

StringTokenizer(String str);

StringTokenizer(String str, String delimiters);

StringTokenizer(String str, String delimiters, Boolean delimAsToken);

delimAsToken –флаг, который символизирует нужно ли возвращать разделитель как лексему.

Некоторые методы:

String nextToken() – возвращает лексему как **String** объект;

boolean hasMoreTokens() – возвращает **true**, если одна или несколько лексем остались в строке;

int countToken() – возвращает число лексем.

Рассмотрим пример:

```
public class StringTokenizerReturnDelimiter{
public static void main(String[] args) {
 StringTokenizer st =
  new
   StringTokenizer("Java|StringTokenizer|Example1",
                                                "|", true);
 while(st.hasMoreTokens()){
   System.out.println(st.nextToken("|"));} } }
```

На экране получим:

Java

|

StringTokenizer

|

Example1

Регулярные выражения

Класс **java.util.regex.Pattern** применяется для определения регулярных выражений (шаблонов), для которых ищется соответствие в строке, файле или другом объекте, представляющем последовательность символов.

Для определения шаблона применяются специальные синтаксические конструкции.

О каждом соответствии можно получить информацию с помощью класса **java.util.regex.Matcher**

Если в строке, проверяемой на соответствие, необходимо, чтобы в какой-либо позиции находился один из символов некоторого символьного набора, то такой набор (класс символов) можно объявить, используя одну из следующих конструкций:

[abc] **a**, **b** или **c**

[^abc] символ, исключая **a**, **b** и **c**

[a-z] символ между **a** и **z**

[a-d[m-p]] либо между **a** и **d**, либо между **m** и **p**

[e-z&&[dem]] **e** либо **m** (конъюнкция)

Кроме стандартных классов символов, существуют предопределенные классы символов:

. любой символ

\d [0-9]

\D [^0-9]

\s символ "пробела“. Эквивалентно

[\t\n\x0B\f\r]

\S [^\s]

\w [a-zA-Z0-9]

\W [^\w]

\p{javaLowerCase} ~ Character.isLowerCase()

\p{javaUpperCase} ~ Character.isUpperCase()

При создании регулярного выражения могут использоваться логические операции:

ab после **a** следует **b**

a|b **a** либо **b**

a) **a**

Скобки, кроме их логического назначения, также используются для выделения групп.

Для определения регулярных выражений недостаточно одних классов символов, т. к. в шаблоне часто нужно указать количество повторений.

Для этого существуют квантификаторы.

a? **a** один раз или ни разу

a* **a** ноль или более раз

a+ **a** один или более раз

a{n} **a** n раз

a{n,} **a** n или более раз

a{n,m} **a** от n до m

Класс **Pattern** используется для простой обработки строк.

Для более сложной обработки строк используется класс **Matcher**

В классе **Pattern** объявлены следующие методы:

- **Pattern compile(String regex)** – возвращает Pattern, который соответствует regex.
- **Matcher matcher(CharSequence input)** – возвращает Matcher,с помощью которого можно находить соответствия в строке input.
- **boolean matches(String regex, CharSequence input)** – проверяет на соответствие строки input шаблону regex.
- **String pattern()** – возвращает строку, соответствующую шаблону.
- **String[ ] split(CharSequence input)** – разбивает строку input,учитывая, что разделителем является шаблон.
- **String[ ] split(CharSequence input, int limit)** – разбивает строку input на не более чем limit частей

С помощью метода **matches()** класса **Pattern** можно проверять на соответствие шаблону целой строки, но если необходимо найти соответствия внутри строки, например, определять участки, которые соответствуют шаблону, то класс **Pattern** не может быть использован.

Для таких операций необходимо использовать класс **Matcher**.

Начальное состояние объекта типа **Matcher** не определено.

Попытка вызвать какой-либо метод класса для извлечения информации о найденном соответствии приведет к возникновению ошибки **IllegalStateException**.

Для того чтобы начать работу с объектом **Matcher**, нужно вызвать один из его методов:

- **boolean matches()** – проверяет, соответствует ли вся строка шаблону;
- **boolean lookingAt()** – пытается найти последовательность символов, начинающуюся с начала строки и соответствующую шаблону;
- **boolean find()**

  boolean find(int start) – пытается найти последовательность символов, соответствующих шаблону, в любом месте строки.

Параметр **start** указывает на начальную позицию поиска.

Иногда необходимо сбросить состояние **Matcher**'а в исходное, для этого применяется метод **reset().**

Для замены всех подпоследовательностей символов, удовлетворяющих шаблону, на заданную строку можно применить метод **replaceAll(String replacement).**

Для того чтобы ограничить поиск границами входной последовательности, применяется метод **region(int start, int end)**, а для получения значения этих границ – **regionEnd()** и **regionStart()**.

С регионами связано несколько методов:

- **Matcher useAnchoringBounds(boolean b)** – если установлен в **true**, то начало и конец региона соответствуют символам **^** и **$** соответственно.
- **boolean hasAnchoringBounds()** – проверяет закрепленность границ.

В регулярном выражении для более удобной обработки входной последовательности применяются группы, которые помогают выделить части найденной подпоследовательности.

В шаблоне они обозначаются скобками “**(**“ и “**)**”. Номера групп начинаются с единицы.

Нулевая группа совпадает со всей найденной подпоследовательностью.

- **int end()** – возвращает индекс последнего символа подпоследовательности, удовлетворяющей шаблону;
- **int end(int group)** – возвращает индекс последнего символа указанной группы;

- **String group(int group)** – возвращает конкретную группу;
- **int groupCount()** – возвращает количество групп;
- **int start()** – возвращает индекс первого символа подпоследовательности, удовлетворяющей шаблону;
- **int start(int group)** – возвращает индекс первого символа указанной группы;
- **boolean hitEnd()** – возвращает истину, если был достигнут конец входной последовательности.

Рассмотрим пример:

```
import java.util.regex.*;
public class DemoRegular {
 public static void main(String[] args) {
 //проверка на соответствие строки шаблону
  Pattern p1 = Pattern.compile("a+y");
  Matcher m1 = p1.matcher("aaay");
 //соответствует строка шаблону
  boolean b = m1.matches();
  System.out.println(b);
 //поиск и выбор подстроки, заданной шаблоном
 String regex ="(\\w+)@(\\w+\\.)(\\w+)(\\.\\w+)*";
 String s = "адреса эл.почты:mymail@a.ua и
                                        rom@b.ua";
```

```
Pattern p2 = Pattern.compile(regex);
Matcher m2 = p2.matcher(s);
while (m2.find()){
    System.out.println("e-mail: " +  m2.group());}
//разбивка строки на подстроки с применением
   шаблона в качестве разделителя
Pattern p3 = Pattern.compile("\\d+\\s?");
String[] words =
          p3.split("java5tiger 77 java6mustang");
for (String word : words){
       System.out.println(word);}
}}
```

На экране получим:

true

e-mail: mymail@a.ua

e-mail: rom@b.ua

java

tiger

java

mustang

Пример. Группы и квантификаторы:

```
public class Groups {
   public static void main(String[] args) {

     myMatches("([a-z]*)([a-z]+)", "abdcxyz");
     myMatches("([a-z]?)([a-z]+)", "abdcxyz");
     myMatches("([a-z]+)([a-z]*)", "abdcxyz");
     myMatches("([a-z]?)([a-z]?)", "abdcxyz");
}
```

```
public static void myMatches(String regex,
                                        String input) {
 Pattern pattern = Pattern.compile(regex);
 Matcher matcher = pattern.matcher(input);
 if(matcher.matches()) {
    System.out.println("First group: "+
                                        matcher.group(1));
    System.out.println("Second group: "+
                                         matcher.group(2));
  } else { System.out.println("nothing");}
 System.out.println();}}
```

На экране получим:

First group: abdcxy

Second group: z

First group: a

Second group: bdcxyz

First group: abdcxyz

Second group:

Nothing

В первом случае к первой группе относятся все возможные символы, но при этом остается минимальное количество символов для второй группы.

Во втором случае для первой группы выбирается наименьшее количество символов, т. к. используется слабое совпадение.

В третьем случае первой группе будет соответствовать вся строка, а для второй не остается ни одного символа, так как вторая группа использует слабое совпадение.

В четвертом случае строка не соответствует регулярному выражению, т. к. для двух групп выбирается наименьшее количество символов.

Интернационализация текста

Класс **java.util.Locale** позволяет учесть особенности региональных представлений алфавита, символов.

Автоматически виртуальная машина использует текущие региональные установки операционной системы, но при необходимости их можно изменять.

Для некоторых стран региональные параметры устанавливаются с помощью констант, например:

Locale.US

Locale.FRANCE.

Для других стран объект **Locale** нужно создавать с помощью конструктора:

Locale myLocale = new Locale(“ua”, “UA”);

Получить доступ к текущему варианту региональных параметров можно следующим образом:

Locale current = Locale.getDefault();

Если, например, в ОС установлен регион «Россия» или в приложении с помощью **new Locale("ru", "RU")**, то следующий код (при выводе результатов выполнения на консоль) позволяет получить информацию о регионе в виде:

current.getCountry();*//код региона*

current.getDisplayCountry();*//название региона*

current.getLanguage();*//код языка региона*

current.getDisplayLanguage();*//название языка региона*

На экране получим:

RU

Россия

ru

русский

Для создания приложений, поддерживающих несколько языков можно использовать взаимодействия классов **java.util.ResourceBundle** и **Locale**.

Класс **ResourceBundle** предназначен в первую очередь для работы с текстовыми файлами свойств (расширение **.properties**).

Каждый объект **ResourceBundle** представляет собой набор объектов соответствующих подтипов, которые разделяют одно и то же базовое имя, к которому можно получить доступ через поле **parent**.

Следующий список показывает возможный набор соответствующих ресурсов с базовым именем **text**.

Символы, следующие за базовым именем, показывают код языка, код страны и тип операционной системы.

Например, файл **text_de_CH.properties** соответствует объекту **Locale**, заданному кодом языка немецкого (**de**) и кодом страны Швейцарии (**CH**).

text.properties

text_ru.properties

text_de_CH.properties

text_en_CA_UNIX.properties

Чтобы выбрать определенный объект **ResourceBundle**, следует вызвать метод **ResourceBundle.getBundle(**параметры**)**. Следующий фрагмент выбирает **text** объекта **ResourceBundle** для объекта **Locale**, который соответствует английскому языку, стране Канаде и платформе UNIX.

```
Locale currentLocale = new Locale("en",
                                  "CA", "UNIX");
ResourceBundle rb =
   ResourceBundle.getBundle("text",
                            currentLocale);
```

В случае если общее определение файла ресурсов не задано, то метод **getBundle()** генерирует исключительную ситуацию **MissingResourceException**.

Чтобы этого не произошло, необходимо обеспечить наличие базового файла ресурсов без суффиксов, а именно: **text.properties** в отличие от частных случаев вида:

text_en_US.properties

Рассмотрим пример:

```
import java.io.IOException;
import java.io.UnsupportedEncodingException;
import java.util.Locale;
import java.util.ResourceBundle;
public class HamletInternational {
 public static void main(String[] args) {
   String country = “”;
   String language = "";
   System.out.println("1 - Английский");
   System.out.println("2 - Украинский");
   System.out.println("Любой символ - Русский");
   char i = 0;
```

```
try {i = (char) System.in.read();}
catch (IOException e1) {e1.printStackTrace();}
switch (i) {
  case '1':
   country = "US";
   language = "EN";
   break;
  case '2':
   country = “UA”;
   language = “UA”;}
Locale current = new Locale(language, country);
ResourceBundle rb =
                    ResourceBundle.getBundle("text", current);
```

```
try {
  String st = rb.getString("str1");
  String s1 =new String(st.getBytes(), "UTF-8");
   System.out.println(s1);
   st = rb.getString("str2");
   String s2 = new String(st.getBytes(), "UTF-8");
   System.out.println(s2);}
catch (UnsupportedEncodingException e) {
   e.printStackTrace();}}}
```

Файл **text_en_US.properties** содержит следующую информацию:

str1 = To be or not to be?

str2 = This is a question.

Файл **text_ua_UA.properties**:

str1 = Бути або не бути?

str2 = Ось у чому питання.

Файл **text.properties**:

str1 = Быть или не быть?

str2 = Вот в чём вопрос.

Интернационализация чисел

Стандарты представления дат и чисел в различных странах могут существенно отличаться.

Чтобы сделать такую информацию конвертируемой в различные региональные стандарты, применяются возможности класса **java.text.NumberFormat**.

Первым делом следует задать или получить текущий объект **Locale** с шаблонами регионального стандарта и создать с его помощью объект форматирования **NumberFormat**. Например:

NumberFormat nf =

NumberFormat.getInstance(

new Locale("RU"));

с конкретными региональными установками или с установленными по умолчанию для приложения:

NumberFormat.getInstance();

Далее для преобразования строки в число и обратно используются методы

Number parse(String source) и

String format(double number)

соответственно

Рассмотрим пример:

```
import java.text.*;
import java.util.Locale;
public class DemoNumberFormat {
 public static void main(String args[]) {
   NumberFormat nfGe =
            NumberFormat.getInstance(Locale.GERMAN);
   NumberFormat nfUs =
                  NumberFormat.getInstance(Locale.US);
   NumberFormat nfFr =
             NumberFormat.getInstance(Locale.FRANCE);
   double iGe=0, iUs=0, iFr =0;
   String str = "1.234,567";
```

```
try{
 //преобразование строки в германский стандарт
  iGe = nfGe.parse(str).doubleValue();
  //преобразование строки в американский стандарт
  iUs = nfUs.parse(str).doubleValue();
   //преобразование строки во французский стандарт
  iFr = nfFr.parse(str).doubleValue();}
catch (ParseException e) {e.printStackTrace();}
   System.out.printf("iGe = %f\niUs = %f\niFr = %f",
                                                    iGe, iUs, iFr);
//преобразование числа из германского в американский
    стандарт
String sUs = nfUs.format(iGe);
//преобразование числа из германского во французский
    стандарт
String sFr = nfFr.format(iGe);
System.out.println("\n" + sUs + "\n" + sFr);}}
```

На экране получим:

iGe = 1234,567000

iUs = 1,234000

iFr = 1,000000

1,234.567

1 234,567

Интернационализация дат

Учитывая исторически сложившиеся способы отображения даты и времени в различных странах и регионах мира, в языке создан механизм поддержки всех национальных особенностей.

Эту задачу решает класс **java.text.DateFormat**.

Процесс получения объекта, отвечающего за обработку регионального стандарта даты, похож на создание объекта, отвечающего за национальные представления чисел, а именно:

DateFormat df = DateFormat.getDateInstance(
DateFormat.MEDIUM, new Locale(“UA”));

или по умолчанию:

DateFormat.getDateInstance();

Константа **DateFormat.MEDIUM** указывает на то, что будут представлены только дата и время без указания часового пояса.

Для получения даты в виде строки для заданного региона используется метод **String format(Date date)** в виде:

String dat = df.format(new Date());

С помощью метода **Date parse(String source)** можно преобразовать переданную в виде строки дату в объектное представление конкретного регионального формата, например:

```
String str = "April 3, 2006";
Date d = df.parse(str);
```

Рассмотрим пример:

```
import java.text.DateFormat;
import java.text.ParseException;
import java.util.*;
```

```
public class DemoDateFormat {
 public static void main(String[] args) {
   DateFormat df = DateFormat.getDateInstance(
                              DateFormat.MEDIUM, Locale.US);
// Если поставить FULL, то будет ошибка
    выполнения, т.к. время не было указано.
   Date d = null;
   String str = "April 3, 2006";
   try {
     d = df.parse(str);
     System.out.println(d);}
   catch (ParseException e) {
     e.printStackTrace();}
```

```
df =DateFormat.getDateInstance(DateFormat.FULL,
                                    new Locale("ru","RU"));
System.out.println(df.format(d));
df=DateFormat.getDateInstance(DateFormat.FULL,
                                    Locale.GERMAN);
System.out.println(df.format(d));
d = new Date();
//загрузка в объект df текущего времени
df = DateFormat.getTimeInstance();
//представление и вывод времени в текущем
   формате дат
System.out.println(df.format(d));}}
```

На экране получим:

Mon Apr 03 00:00:00 EEST 2006

3 Апрель 2006 г.

Montag, 3. April 2006

05:45:16